Unidad II: Programación orientada a objetos básica

Lic. Ronaldo Armando Canizales Turcios

Departamento de Electrónica e Informática, UCA

Ciclo virtual 01-2021

Agenda

1. **Herencia**
   - Polimorfismo
   - Herencia de métodos
   - Herencia de atributos
   - Ejemplo - cuentas chequeras
   - Herencia y constructores
   - Abstract - forzando la herencia

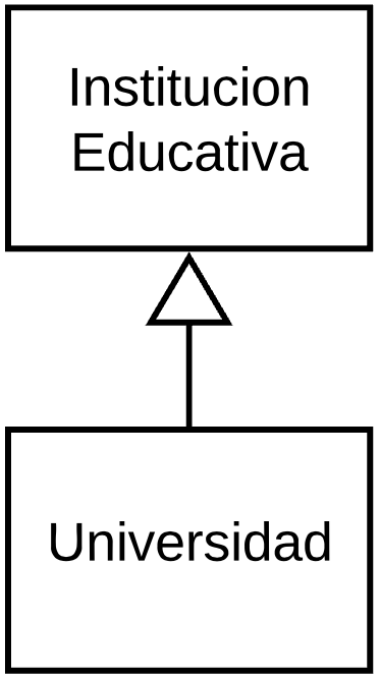
1. Herencia

Herencia

Es un mecanismo utilizado en la programación orientada a objetos para *extender* (hacer más grande, más poderosa) una clase que ya existe e inclusive ya se utiliza para realizar tareas.

Si se necesita implementar una clase nueva pero ya se ha creado con anterioridad una clase que está ligada a un concepto más general, entonces la nueva clase puede heredar de la clase existente. Por ejemplo suponga que usted ya creó una clase llamada **InstitucionEducativa** y entonces le piden implementar otra clase llamada **Universidad**, entonces usted puede tomar todos los elementos de la clase **InstitucionEducativa** y hacer que sean heredados por la nueva clase a crear.

1. Herencia



Herencia de una clase en C#

```
class InstitucionEducativa {
    atributos del padre
    métodos del padre
}
class Universidad : InstitucionEducativa {
    nuevos atributos
    nuevos métodos
}
```

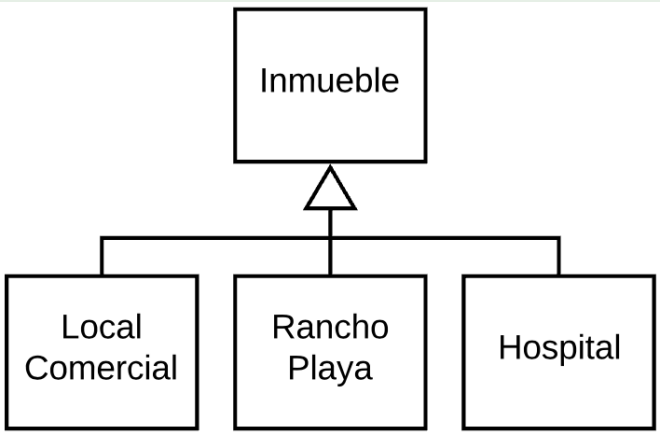
Los nuevos métodos y nuevos atributos sólo los posee la clase Universidad. La herencia permite crear clases más personalizadas. Que una clase hija herede de otra permite reutilizar la funcionalidad **base** (componentes de la clase padre) en lugar de programarla desde cero.

1. Herencia

Una clase puede heredar solamente de una clase, pero ella puede ser heredada por varias; como analogía las personas tienen sólo un padre biológico, pero un padre biológico puede tener varios hijos. Por ejemplo un **Inmueble** es un caso general de **LocalComercial, RanchoPlaya** y **Hospital**.

Una clase padre con varias clases hijas en C#

```
class Inmueble {
    public string direccion;
    public double area;
}
```




```
class LocalComercial : Inmueble {
    public string nombreNegocio;
}
class RanchoPlaya : Inmueble {
    public int cantidadPiscinas;
}
class Hospital : Inmueble {
    public int cantidadCamas;
}
```

1. Herencia

Es hora de introducir terminología:

Súper-clase

A la clase más general que forma la base de la herencia (por ejemplo Inmueble) se le conoce como *súper-clase* (también *clase base* o *clase padre*).

Sub-clases

Mientras que a la clase más específica, que hereda de la súper clase (por ejemplo LocalComercial, RanchoPlaya u Hospital) se les conoce como *sub-clases* (también es aceptado *clases derivadas)*.

1. Herencia

Las clases **LocalComercial, RanchoPlaya** y **Hospital** tienen todos los atributos y métodos de la clase **Inmueble** además de los propios de cada uno.

```
static void Main(String[ ] args) {
    Inmueble unInmueble = new Inmueble();
    unInmueble.direccion = "San Miguel";
    unInmueble.area = 150;

    LocalComercial unLocal = new LocalComercial();
    unLocal.direccion = "San Salvador";
    unLocal.area = 100;
    unLocal.nombreNegocio = "Venta de comida";
}
```

Versión 1

1. Herencia

Reutilización de código

Una de las mayores ventajas de la herencia es la **reutilización de código**. Al heredar de una clase existente, el programador ya no tiene que dedicar tiempo y esfuerzo en el diseño e implementación de la funcionalidad base.

Teniendo en cuenta que una sub-clase tiene más atributos, más métodos, es una versión extendida de la súper-clase; entonces tal vez te preguntes: ¿porqué una sub-clase se llama así si parece ser mejor que la súper-clase?

La terminología *súper/sub* proviene de la teoría de conjuntos, es decir, los locales comerciales son un **subconjunto** de los Inmuebles. Hay que tener en cuenta que los objetos más especializados (y con más capacidades) siempre serán subconjuntos.

1.1 Herencia: Polimorfismo

Las variables de tipo objeto son polimórficas

Esto significa que una variable de la clase X puede referirse a un objeto tanto de la clase X como de cualquiera de sus sub-clases.
Por ejemplo: una variable de la clase Inmueble puede contener objetos de las clases Inmueble, LocalComercial, RanchoPlaya u Hospital.

```
static void Main(String[ ] args) {
    LocalComercial unLocal = new LocalComercial();
    mostrarDatos(unLocal);
}
static void mostrarDatos(Inmueble unInmueble) {
    sout("Dirección: " + unInmueble.direccion);
    sout("Área: " + unInmueble.area + " m**2");
}
```

Versión 2

1.1 Herencia: Polimorfismo

Error común: es inválido en el sentido inverso

Una variable de la clase LocalComercial puede referirse a:

- Objetos de la clase LocalComercial.
- Objetos de sub-clases de LocalComercial (este ejemplo no hay).
- NO a objetos de la clase Inmueble (padre).
- NO a objetos de las clases Rancho ni Hospital ("hermanas").

Uso incorrecto de parámetros polimórficos

```
static void Main(String[ ] args) {
    Inmueble unInmueble = new Inmueble();
    mostrarDatos(unInmueble);
}
static void mostrarDatos(LocalComercial unLocal) {
    sout("Negocio: " + unLocal.nombreNegocio);
}
```

Erróneo

1.2 Herencia: Herencia de métodos

Al definir nuevos métodos en las sub-clases existen tres posibilidades:

1) Sobrescribir métodos de la súper-clase:

Si se implementa en la sub-clase un método con la misma *signatura* que un método de la súper-clase (es decir, con el mismo nombre y los mismos parámetros), entonces se sobrescribe el método de la súper-clase. Esto afecta sólo a la sub-clase, el método de la súper-clase queda intacto.

Cada vez que dicho método sea invocado desde un objeto de la sub-clase, entonces el método que se ejecute será el nuevo (el que se implementó en la sub-clase, el método que sobre-escribió) y no el original (el implementado en la súper-clase).

1.2 Herencia: Herencia de métodos

2) Heredar métodos de la súper-clase:

Si NO se implementa explícitamente un método en la sub-clase que sobrescriba dicho método de la súper-clase (es decir, si no se hace nada) entonces se hereda de manera automática.

Cada vez que dicho método sea invocado desde un objeto de la sub-clase, entonces el método que se ejecute será el original (el implementado en la súper-clase).

3) Definir nuevos métodos:

Si se implementa un método en la sub-clase que NO exista en la súper-clase entonces dicho método puede aplicarse únicamente a objetos de la sub-clase.
La súper-clase NO tendrá acceso a ese método recién implementado.

1.3 Herencia: Herencia de atributos

La situación para atributos es un poco distinta ya que **no se pueden sobrescribir los atributos** (pero sí las propiedades, los getters y setters). Cuando se definen atributos en una sub-clase existen únicamente dos escenarios:

1) Heredar atributos de la súper-clase:

Todos los atributos de la súper-clase son heredados de manera automática.

2) Definir nuevos atributos:

Cualquier atributo nuevo que se defina en una sub-clase existirá únicamente en objetos de dicha sub-clase.

1.3 Herencia: Herencia de atributos

- La palabra reservada **virtual** permite la posibilidad de que un método sea sobreescrito en una subclase.
- Por defecto, los métodos son **no virtuales**. Hay que especificarlo.
- Para sobreescribir se utiliza la palabra reservada **override**.

`virtual` te permite que una propiedad y método pueda ser sobreescrita por una clase que herede de ella.

Los casos en que la he utilizado han sido mas cuando necesito modificar el comportamiento por defecto de la clase como por ejemplo que al momento de obtener el nombre lo convierta en mayúscula ya que la clase principal no lo hace.

```
public class Persona
{
    public virtual string Nombre { get; set; }
}

public Cliente : Persona
{
    public override string Nombre {
        get { base.Nombre.ToUpper(); }
        set { base.Nombre = value; }
    }
}
```

1.3 Herencia: Herencia de atributos

¿Qué sucede si se define en una sub-clase un atributo con el mismo nombre que un atributo de su súper-clase? Por ejemplo, ¿qué sucedería si se define el atributo **dirección** en la clase RanchoPlaya (recordar que ya existe un atributo llamado **dirección** en la clase Inmueble)?

Shadowing

Lo que sucedería es que cada objeto de la clase RanchoPlaya tendría dos atributos llamados dirección. El nuevo atributo hace que el viejo "se esconda" (*shadowing*), es decir que se tendría acceso únicamente al nuevo atributo (el declarado en la sub-clase) mientras que el atributo declarado en la súper-clase todavía está presente pero no puede ser accedido por los métodos de la clase RanchoPlaya. La práctica de crear y utilizar *atributos sombra* puede ser una fuente de confusión, por lo tanto se recomienda evitarla.

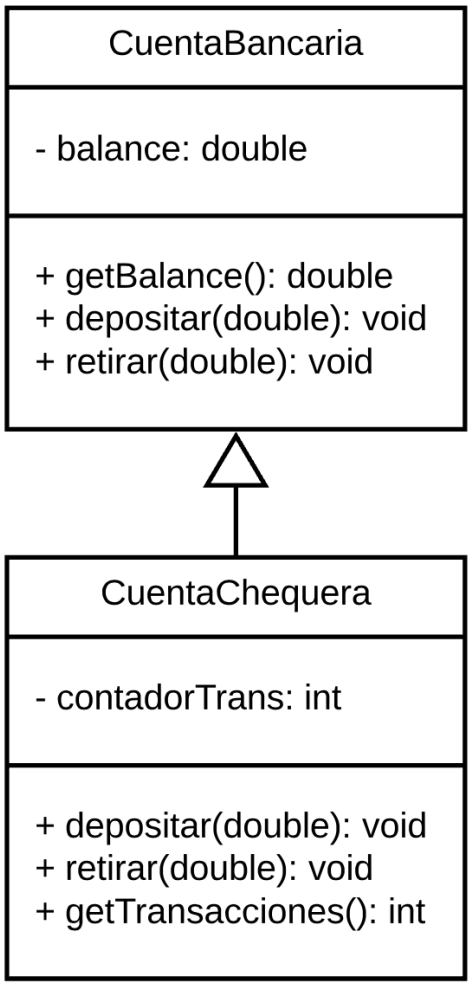
1.4 Herencia: Ejemplo - cuentas chequeras

Ejemplo: Cuentas de banco y cuentas chequeras

Implementemos las siguientes clases:

- **Cuenta Bancaria:** clase que representa cualquier cuenta bancaria en general. Tiene un único atributo, su balance, y cuenta con tres métodos: consultar balance, depositar y retirar.
- **Cuenta Chequera:** clase que representa un tipo particular de cuenta, es un sub-conjunto de cuenta bancaria. Hace exactamente lo mismo que la cuenta bancaria excepto que tiene un atributo llamado **contador de transacciones** que deberá incrementarse cada vez que un usuario realice un depósito o un retiro de dinero.

1.4 Herencia: Ejemplo - cuentas chequeras




```
public class CuentaBancaria {
    private double balance;

    public double getBalance() {...}
    public virtual void depositar(double cantidad) {...}
    public virtual void retirar(double cantidad) {...}
}
```

```
public class CuentaChequera : CuentaBancaria {
    private int contadorTransacciones;

    public overridevoid depositar(double cantidad) {...}
    public override void retirar(double cantidad) {...}
    public int getTransacciones() {...}
}
```

1.4 Herencia: Ejemplo - cuentas chequeras

Cada objeto de la clase CuentaChequera tiene los siguientes **atributos**:

- balance (heredado de CuentaBancaria)
- contadorTransacciones (nuevo en CuentaChequera)

También, cada objeto de la clase CuentaChequera tiene los **métodos**:

- getBalance() (heredado de CuentaBancaria)
- depositar(double) (sobrescribe el método de CuentaBancaria)
- retirar(double) (sobrescribe el método de CuentaBancaria)
- getTransacciones() (nuevo en CuentaChequera)

1.4 Herencia: Ejemplo - cuentas chequeras

Al momento de implementar los métodos de la clase CuentaChequera nos topamos con el siguiente inconveniente:

Una sub-clase NO tiene acceso a los atributos **privados** de su súper-clase.

Código erróneo

```
public class CuentaChequera : CuentaBancaria {
    ...
    public override void depositar(double cantidad) {
        contadorTransacciones++;
        balance = balance + cantidad; Error...
    }                          No se tiene acceso porque es privado
    ...                            ¡Solo la clase padre tiene acceso!
}
```

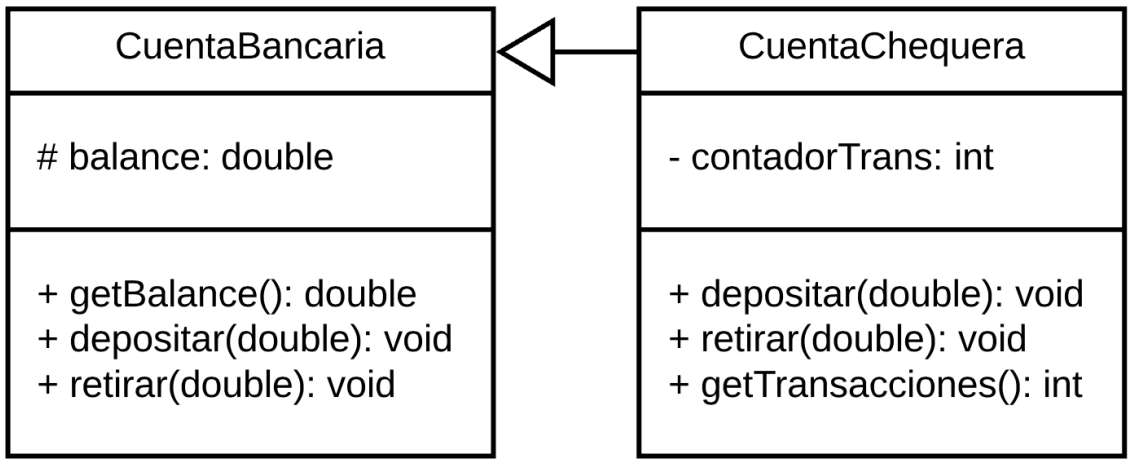
1.4 Herencia: Ejemplo - cuentas chequeras

¿Cuáles alternativas hay para solventar este inconveniente?

Alternativa 1: cambiar private por protected

El especificador de acceso **protected** hace a un método o a un atributo accesible dentro de la misma clase y en cualquier instancia (objeto) de sus clases derivadas (sub-clases).

Es importante recalcar la palabra *instancia*, ya que eso implica que un atributo o método protected no podrá ser accedido en métodos estáticos.



Alternativa 1

1.4 Herencia: Ejemplo - cuentas chequeras

Alternativa 2: usar la interfaz pública de la clase padre

¿Habrá alguna forma de modificar el atributo balance mediante la interfaz pública de la clase CuentaBancaria? Existe una respuesta idónea: el método **depositar** de la clase CuentaBancaria (súper-clase).

Código erróneo

```
public class CuentaChequera : CuentaBancaria {
    ...
    public override void depositar(double cantidad) {
        contadorTransacciones++;
        depositar(cantidad); Ups, sigue habiendo error...
    }          Se está llamando al método de la clase actual (recursión)
    ...          Esto no es lo que queremos, sino llamar a la clase padre
}
```

1.4 Herencia: Ejemplo - cuentas chequeras

Palabra reservada base

Un momento, al hacer lo anterior se está accediendo al **método depositar de la clase CuentaChequera** pero lo que se desea es acceder al método depositar de la clase CuentaBancaria (súper-clase), ¿cómo lograrlo en Java? mediante la palabra reservada **super**.

```
public class CuentaChequera : CuentaBancaria {
    ...
    public override void depositar(double cantidad) {
        contadorTransacciones++;
        base.depositar(cantidad); Ahora no hay error.
    }   Estamos llamando al método de la clase padre.
    ...
}
```

Alternativa 2

1.5 Herencia: Herencia y constructores

En C#, se puede utilizar en una sub-clase la palabra reservada **base** seguida de paréntesis (y parámetros de construcción) para llamar un constructor de la súper-clase.

```
public class CuentaBancaria {
    public CuentaBancaria(double balanceInicial) {
        balance = balanceInicial;
    }
}
public class CuentaChequera extends CuentaBancaria {
    public CuentaChequera(double balanceInicial){
        super(balanceInicial);
        contadorTransacciones = 0;
    }
}
```

Continuación de Alternativa 2

1.6 Herencia: Abstract - forzando la herencia

Palabra reservada abstract

Permite crear clases y métodos que están **incompletos** y deben ser implementados en una clase derivada (sub-clase).

Caract. de las clases abstractas:

- Una clase abstracta NO puede ser instanciada (crear objetos).
- Puede contener métodos abstractos.
- Una clase B (no abstracta) que herede de una clase A (abstracta) debe proveer implementaciones de todos los métodos abstractos que ha heredado.

1.6 Herencia: Abstract - forzando la herencia

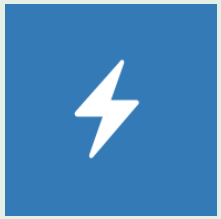
Características de los métodos abstractos:

- Un método abstracto sólo puede ser declarado dentro de una clase abstracta.
- Los métodos abstractos no proveen implementación, por ello no tienen cuerpo (no tienen llaves de apertura o cierre, ni instrucciones dentro) sin embargo, deben terminar con un punto y coma.

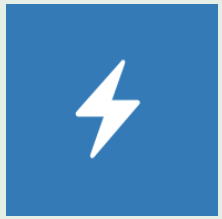
  ¿Dónde se digita su implementación? Debe sobrescribirse dicho método en una clase no abstracta que lo herede (en una sub-clase).

Sintaxis de método abstracto

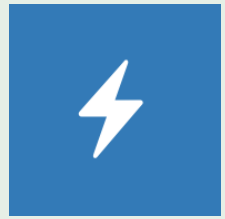
public abstract void nombreMetodo();



Ejemplo 1



Ejemplo 2



Ejemplo 3

Unidad II: Programación orientada a objetos básica

Lic. Ronaldo Armando Canizales Turcios

Departamento de Electrónica e Informática, UCA

Ciclo virtual 01-2021